Traffic Measurement

Introduzione

La misura del traffico di rete rende la rete stessa più osservabile. L'ingegneria del traffico invece la rende più controllabile.

Gli strumenti di base della misura del traffico sono:

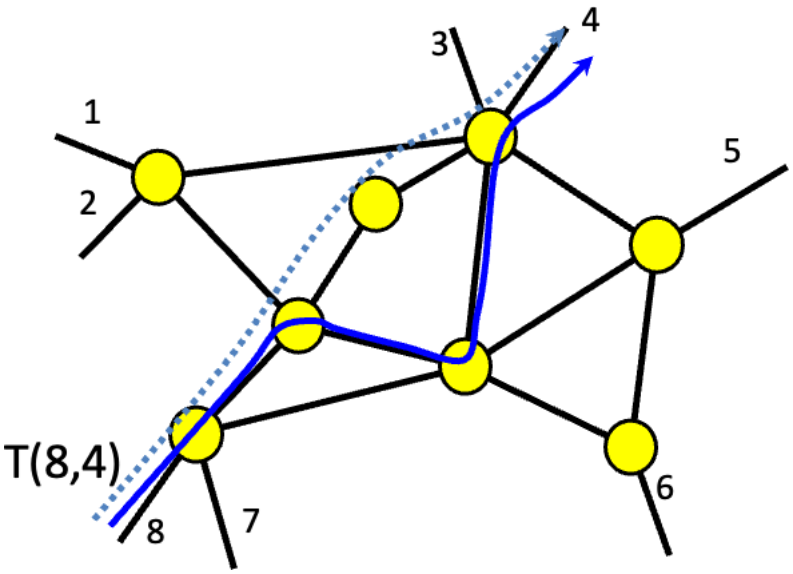
- Conteggio dei pacchetti
- Conteggio degli ottetti

Questi due strumenti sono usati per:

- L'identificazione dei flow di traffico più significativi per il *capacity planning*
- La rilevazione di possibili attacchi
- La fatturazione per un servizio di rete
- La verifica dei *Service Level Agreement*

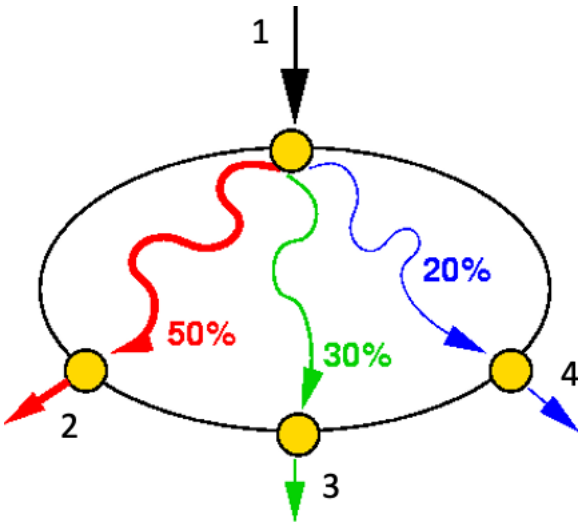
Traffic Matrix

Una *Traffic Matrix* restituisce, per ogni punto di ingresso i ed ogni punto di uscita j di una rete, il volume di traffico $T(i, j)$ tra i e j in un dato intervallo di tempo.



[!warning] Una Traffic Matrix non considera i percorsi interni alla rete.

Consideriamo ora questa matrice:



Possiamo calcolare $T^{\text{in}}(1)$ come:

$$T^{\text{in}}(1) = T(1,2) + T(1,3) + T(1,4)$$

Inoltre possiamo anche misurare il traffico a ciascun nodo di uscita. Non possiamo però misurare $T(1,2)$: dobbiamo calcolarne una stima.

Stima del traffico

Si consideri una rete con N link di ingresso o uscita.

La matrice di traffico di questa rete è $T(i,j)$, con $1 \leq i \leq N$ e $1 \leq j \leq N$. Altre proprietà di questa matrice sono:

- $T(i,i) = 0, \ \forall i$
- $T^{\text{in}}(i) = \sum_{j=1}^{N} T(i,j)$
- $T^{\text{out}}(j) = \sum_{i=1}^{N} T(i,j)$
- $\sum_{i=1}^{N} T^{\text{in}}(i) = \sum_{j=1}^{N} T^{\text{out}}(j)$

Giungiamo quindi ad una matrice di questo tipo per $N = 3$:

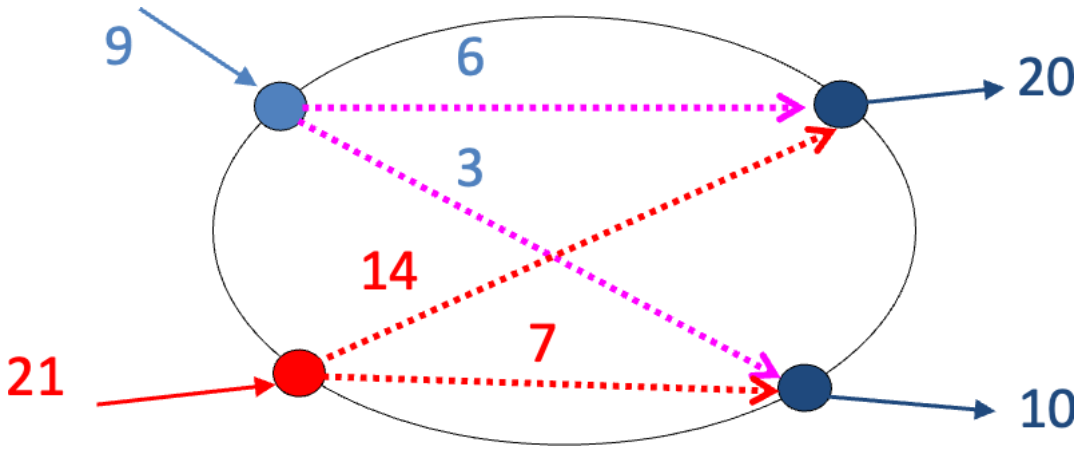
$$T = \begin{bmatrix} 0 & T(1,2) & T(1,3) \\ T(2,1) & 0 & T(2,3) \\ T(3,1) & T(3,2) & 0 \end{bmatrix}$$

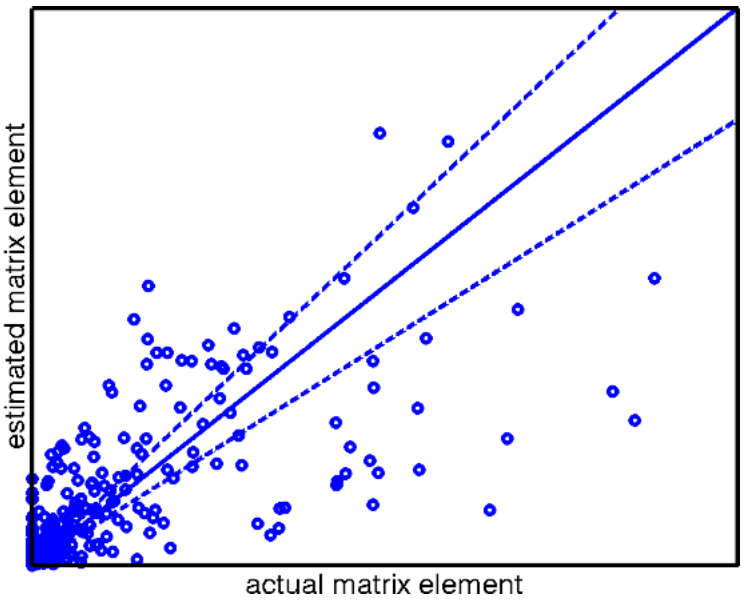
Abbiamo $2N + 1$ equazioni ed $N(N-1)$ incognite, perciò esistono diverse soluzioni del sistema di equazioni associato alla matrice di traffico.

Per poter stimare il traffico, dobbiamo fare ancora qualche assunzione:

[!abstract] Gravity Model Il traffico tra ogni coppia di link è proporzionale al traffico sui link stessi:

$$T(i,j) = \frac{T^{\text{in}}(i) T^{\text{out}}(j)}{\left(\sum_{j=1}^{N} T^{\text{out}}(j)\right) - T^{\text{out}}(i)}$$





Questo modello è semplice ma non molto accurato e può essere rifinito aggiungendo informazioni su percorsi di routing, misure su link interni ed altro.

Exact Counting

Lo strumento base delle misure del traffico è il conteggio di pacchetti e byte.

Per contare i pacchetti in ingresso ad un nodo però servono tanti contatori di grandi dimensioni (anche 64 bit ciascuno) e, per ogni pacchetto, più di un contatore dev'essere aggiornato. Questo richiede della memoria molto veloce.

> [!question] In che modo è possibile risparmiare sulla memoria veloce?
>
> [!example] Idea Possiamo, per esempio, tenere versioni "corte" dei contatori in banchi di memoria veloce, mentre i contatori completi vengono salvati nella memoria lenta.

Periodicamente, viene selezionato un contatore dalla memoria veloce; il suo valore viene aggiunto al contatore lento ed il contatore veloce viene resettato.

La complessità di questo algoritmo non dovrebbe dipendere dal numero di contatori.

Per assicurarci la correttezza di questo algoritmo, ogni contatore dev'essere salvato prima che vada in overflow.

> [!question] Quanti bit servono ai contatori veloci per assicurarsi che non vadano in overflow?

Ipotizziamo che i contatori corti siano di c bit, mentre quelli lunghi siano di M bit, con $M > c$. Inoltre, ipotizziamo di avere N contatori.

Ad ogni arrivo di pacchetto, aggiorniamo un contatore nella memoria veloce. Se la memoria lenta è b volte più lenta di quella veloce, ad ogni b pacchetti in ingresso, salviamo e resettiamo un contatore veloce.

> [!question] Quale contatore salviamo?
>
> [!abstract] Largest Count First (LCF) Salviamo e resettiamo il contatore con il valore maggiore. > [!note] Teorema > Con LCF, non c'è overflow nei contatori se: >
>
> $$c \approx \log\log(bN)$$
>
> [!failure] Problema L'algoritmo per trovare il numero più grande ha complessità che dipende da N.

Proviamo a semplificare questo algoritmo:

> [!abstract] Simplified LCF Sia S l'array di contatori nella memoria veloce e sia $S[i]$ il valore del contatore di indice i. Sia j l'indice del contatore con valore massimo aggiornato nell'ultimo ciclo di b pacchetti. Ogni b pacchetti, se $S[j] \geq b$, allora salva e resetta il contatore di indice j; altrimenti salva e resetta qualsiasi altro contatore con valore almeno b. > [!note] È necessario un c poco più grande rispetto a LCF standard.

Randomized Counting

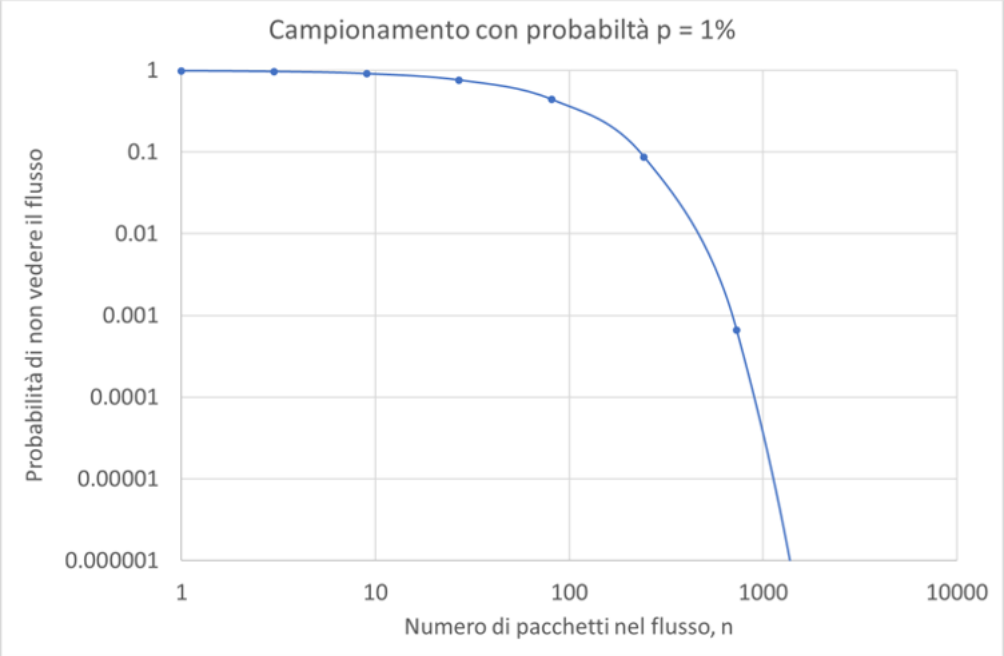
Proviamo con un approccio diverso: quando arriva un pacchetto, un contatore viene incrementato con probabilità $p = 1/c$. In media, quindi, un contatore viene incrementato ogni c pacchetti.

Se il contatore incrementato ha valore x, possiamo stimare che il numero di pacchetti ricevuti è xc.

> [!note] La precisione della stima dipende dalla dimensione n del flusso: - Quando $n \gg c$, la precisione è buona - Quando $n \ll c$, la precisione è bassa

Il Randomized Counting funziona bene quando vogliamo misurare flussi di dimensioni significative: la misura di eventi poco frequenti è inaccurata.

> [!question] Qual è la probabilità di mancare un flusso di n pacchetti? Sia n la dimensione del flusso in pacchetti e sia X la variabile aleatoria *valore del contatore*. X segue una distribuzione binomiale $B(n, p)$:
>
> $$P[X = 0] = (1 - p)^n$$



Calcoliamo una stima del numero di pacchetti in un flusso: sia n la dimensione del flusso in pacchetti e X la variabile aleatoria *valore del contatore*. Come detto prima, X segue una distribuzione binomiale $B(n, p)$, perciò:

$$\begin{aligned} E[X] &= np \\ \mathrm{Var}[X] &= np(1 - p) \end{aligned}$$

Lo stimatore per il numero di pacchetti è $\hat{n} = X/p$. Il valore atteso dello stimatore è:

$$\begin{aligned} E[\hat{n}] &= E\left[\frac{X}{p}\right] \\ &= \frac{1}{p} E[X] \\ &= n \end{aligned}$$

La sua varianza, invece, è:

$$\begin{aligned}\mathrm{Var}[\hat{n}] &= \mathrm{Var}\left[\frac{X}{p}\right] \\ &= \frac{1}{p^2}\mathrm{Var}[X] \\ &= \frac{np(1-p)}{p^2} \\ &= \frac{n(1-p)}{p}\end{aligned}$$

La varianza dell'errore relativo è:

$$\begin{aligned}\mathrm{Var}\left[\frac{n-\hat{n}}{n}\right] &= \mathrm{Var}\left[\frac{\hat{n}}{n}\right] \\ &= \frac{1}{n^2}\mathrm{Var}[\hat{n}] \\ &= \frac{1-p}{np}\end{aligned}$$

La varianza dell'errore relativo è *piccola per gli elefanti e grande per i topi*[1].

Count-Min Sketches

> [!important] Attenzione Il metodo di stima probabilistica che utilizza i Count-Min Sketches è applicabile a diversi casi d'uso, ma noi lo vedremo nel dettaglio per un solo scopo.

Rilevazione degli heavy-hitter

Ipotizziamo di voler identificare i flussi più pesanti in una rete, ossia quelli che stanno occupando la maggior parte delle risorse della rete stessa.

Per farlo, possiamo analizzare gli header di ogni pacchetto in ingresso nella rete, estrarne alcune informazioni ed elaborarle tramite una funzione di hash, in modo da creare un identificatore univoco per una classe di pacchetti, che useremo per identificare i flussi in ingresso.

Dato che la funzione hash usata per svolgere questa operazione trasforma una stringa di bit di dimensione arbitraria in una diversa stringa di bit di dimensione fissa:

$$h : \{0,1\}^* \to \{0,1\}^l$$

possiamo interpretarne il risultato come un numero intero nell'intervallo $[0, 2^l - 1]$.

Calcolato questo numero intero, possiamo utilizzarlo come indice di un array che contiene numeri interi. In questo modo, incrementiamo effettivamente un contatore che tiene traccia del numero di pacchetti appartenenti ad uno stesso flusso in ingresso alla rete.

Possiamo poi stabilire una certa soglia T tale per cui:

- Se il contatore è inferiore a T, allora il flusso *non è* un heavy hitter
- Se il contatore supera T, allora il flusso *è* un heavy hitter

Il problema nell'uso di questo approccio è la grande quantità di falsi positivi indotta dalle collisioni causate dalla funzione di hash, che sono inevitabili. È possibile infatti che un numero significativo di flussi piccoli collida in un unico contatore, facendo così credere che ci sia un flusso heavy-hitter.

Per risolvere questo problema, è sufficiente aumentare il numero di funzioni hash utilizzate ed il numero di array di contatori. In questo modo, la probabilità di collisioni su tutte le eventuali k funzioni hash diminuisce drasticamente.

Questo metodo appena descritto si definisce **Threshold Aggregation** e la struttura dati che abbiamo finora descritto come semplici array di interi viene definita **Count-Min Sketch**.

[1] Si dice *elefante* un flusso con vita molto lunga, mentre *topo* un flusso con vita molto corta.

Un Count-Min Sketch non è altro che una matrice di k righe per 2^l colonne, dove k è, come detto prima, il numero di funzioni hash usate ed l è la lunghezza in bit dell'output delle varie funzioni di hash[2].

Le primitive associate ad un Count-Min Sketch sono fondamentalmente due:

Primitiva	Argomenti	Descrizione
`update(x, c)`	`x`: elementi di cui calcolare l'hash; `c`: quantità di cui incrementare il contatore	Questa primitiva aggiorna i contatori relativi alle k funzioni di hash calcolate sul pacchetto `x`.
`estimate(x)`	`x`: elementi di cui calcolare l'hash	Questa primitiva stima il traffico occupato dal flusso identificato da `x`. Ritorna il minimo valore tra i contatori identificati dall'hash di `x`.

Possiamo stimare che, in media, l'errore portato dal calcolo della stima di traffico di un flusso è N/m, dove N è il numero di elementi salvati nel Count-Min Sketch ed m è il numero di colonne del Count-Min Sketch. L'errore medio è il *rumore di fondo* causato dalle collisioni delle funzioni di hash, che aggregano flussi di diversa entità.

Un calcolo più accurato ci porta a dire che la probabilità che il rumore di fondo sia minore di $2N/m$ è di $1/2$.

Dato che la primitiva `estimate(x)` ci restituisce il minimo valore tra i contatori di una stessa classe di pacchetti, scegliamo in automatico la cella con il minimo rumore di fondo. Possiamo perciò dire che la probabilità che il rumore di fondo sia meno di $2N/m$ sale esponenzialmente con il numero di funzioni di hash k usate:

$$P\left(\text{rumore minimo} < \frac{2N}{m}\right) = 1 - \frac{1}{2^k}$$

In conclusione, se $\hat{w}$ è la stima restituita da `estimate(x)`, possiamo dire che:

$$\hat{w} = w + \frac{2N}{m}$$

dove w è il numero esatto di pacchetti ricevuti da un certo flusso.

> [!warning] Attenzione Aumentare il numero di funzioni hash diminuisce notevolmente la possibilità di incontrare falsi positivi, ma aumenta anche la quantità di memoria occupata dal Count-Min Sketch, dato che il suo numero di righe è pari al numero di funzioni hash usate.

Probabilistic Counting

> [!warning] Attenzione Anche questo algoritmo ha molteplici casi d'uso, ma noi lo vediamo solo in un contesto specifico.

Questo algoritmo serve per contare quanti flussi unici sono stati osservati nella vita della rete.

Questo algoritmo può rispondere ad una domanda del tipo:

> [!question] Quante connessioni TCP distinte sono transitate da un dato router nell'ultima ora?

Questo algoritmo è alla base di diversi meccanismi di identificazione di anomalie, come la rilevazione di attacchi DoS.

L'algoritmo che utilizziamo per raggiungere questo scopo è l'algoritmo di **Flajolet-Martin**, sviluppato nel 1984.

Anche in questo caso, otterremo numeri approssimati, poiché perdiamo accuratezza per risparmiare memoria.

Come per l'identificazione degli heavy-hitters tramite Count-Min Sketches, ci basiamo su delle funzioni di hash per elaborare l'header dei pacchetti in ingresso, ma con lo scopo di identificare pattern non comuni. Maggiore è la probabilità di vedere pattern non comuni, maggiore è il numero di flow distinti osservati[3].

Per un singolo valore hash, sia K la posizione del primo uno nell'hash[4]. K è una variabile aleatoria geometrica con $p = 1/2$. Perciò:

[2] Le funzioni di hash applicate nella Threshold Aggregation devono avere tutte pari lunghezza dell'output.
[3] Questo succede perché, salvo collisioni, ogni flow diverso darà luogo ad un output diverso della funzione di hash.
[4] Questo numero è pari al numero di zeri in testa all'hash, più uno.

$$P[K < k] = 1 - \frac{1}{2^k}$$

Dato che abbiamo un numero di hash per ciascun flusso unico, la posizione dell'uno più a destra che osserviamo su w flussi è:

$$P[X < x] = \big(P[K < x]\big)^w = \left(1 - \frac{1}{2^x}\right)^w$$

Maggiore è il numero di flussi w, maggiore sarà la probabilità che l'uno più a destra sia molto a destra nel valore di hash.

Si può calcolare che $P[X = x]$ è massimo a circa $x = \log_2(w)$. Questo significa che spesso il numero di zeri iniziali degli hash di ciascun flusso è **pari al logaritmo in base due del numero dei flussi incontrati**.

Questa osservazione è alla base dell'algoritmo di Flajolet-Martin:

1. Calcolare il flow ID di ciascun flusso tramite delle funzioni di hash
2. Per ciascun flow ID, troviamo il primo uno nella stringa di bit
3. Se x è la posizione dell'uno più a destra su tutti gli hash osservati, la stima del numero di flussi w è pari a:

$$\hat{w} = \frac{2^x}{\varphi},\ \varphi = 0.77351$$

[!warning] Attenzione La varianza dello stimatore $\hat{w}$ è molto alta. Infatti, se un solo flusso ha il primo uno molto più a destra di tutti gli altri, è possibile sbagliare di qualche ordine di grandezza la stima del numero di flussi nella rete. Per attenuare questo errore, è sufficiente applicare la stessa stima con diverse funzioni di hash e fare la media delle varie stime ottenute.

[!abstract] Algoritmo di Flajolet-Martin Completo Si considerino m funzioni di hash e sia x_i l'uno più a destra del risultato dell'i-esima funzione di hash. Si calcoli la media della posizione più a destra tra gli output delle varie funzioni di hash come:

$$A = \frac{1}{m}\sum_{i=1}^{m} x_i$$

Allora, la stima per il numero di flussi unici in una rete è pari a:

$$\hat{w} = \frac{2^A}{\varphi},\ \varphi = 0.77351$$

[!warning] Attenzione Aumentare il numero m di funzioni hash abbassa la varianza dello stimatore, ma aumenta il numero di risorse richiesto per calcolarle.

Hash-based sampling

Trajectory Sampling

Ipotizziamo di voler seguire la traiettoria di un pacchetto in una rete. Per farlo, è necessario campionare il pacchetto in ogni router in cui esso passa.

[!question] Come possiamo forzare i router a prendere delle decisioni coordinate per il campionamento dei pacchetti?

Possiamo, ad esempio, estrarre una stringa di bit da ciascun pacchetto da campi di esso che non cambiano ad ogni hop e calcolarne un digest.

[!question] Quali campi usare? Gli indirizzi IP di sorgente e destinazione ed il payload del pacchetto non cambiano tra hop, ma rimangono invariati fino a destinazione. I campi di TTL e indirizzi MAC di sorgente e destinazione, invece, cambiano ad ogni switch attraversato dal pacchetto.

Chiamiamo questa stringa m. Usiamo una funzione di hash h, identica in tutti i router, per calcolarne il digest. Se $h : \{0,1\}^* \to [0, R)$, allora il digest della stringa m è un numero $0 \leq h(m) < R$.

Scegliamo ora una soglia $B < R$. Se $h(m) < B$, allora il pacchetto viene contato.

Se ipotizziamo che l'output della funzione h sia una variabile aleatoria uniforme, un dato pacchetto viene contato in ogni router con probabilità B/R.

Per campionare il pacchetto, possiamo sia salvare l'intero pacchetto, sia salvarne il digest. Nel caso del salvataggio di digest, è necessario usare una seconda funzione di hash. Questo perché, dato che tutti i pacchetti campionati vengono elaborati con la stessa funzione di hash h e salvati solo se sotto la stessa soglia B, per evitare collisioni e poter distinguere meglio i pacchetti campionati, è necessario usare una seconda funzione di hash g prima del salvataggio del digest del pacchetto.

Bloom Filter

> [!warning] Attenzione Il Bloom Filter ha diverse applicazioni[5], ma noi lo vediamo applicato ad un problema specifico.

Ipotizziamo che ci sia un pacchetto in rete che è stato identificato come malevolo. Il nostro obiettivo è capire da dove sia arrivato questo pacchetto e come abbia fatto ad entrare non essendo autorizzato.

Guardare l'IP sorgente non è utile, perché può essere cambiato per mascherare la vera origine del pacchetto[6]. Inoltre, gli indirizzi falsificati non possono essere filtrati, perché:

- Filtrare IP che non appartengono alla propria rete è possibile solo se si è al bordo della rete, come nel caso degli operatori telefonici
- Filtrare tutti i pacchetti che arrivano dallo stesso percorso fatto dal pacchetto richiede che il routing sia simmetrico[7]

Possiamo provare a risalire fino alla sorgente del pacchetto malevolo *registrando* il pacchetto in ogni nodo in cui passa e, nel momento in cui arriva alla destinazione, chiedere ai nodi vicini da dove sia passato.

La problematica relativa a questo approccio è lo spazio su disco occupato dai log: salvare ogni pacchetto che transita in un nodo richiede moltissimo spazio.

Serve quindi un meccanismo che permetta in modo veloce e compatto di testare se un pacchetto è transitato per un nodo.

Qui entra in gioco il **filtro di Bloom**, algoritmo che permette di verificare l'*appartenenza ad un insieme* in modo rapido, senza occupare molto spazio.

Un filtro di Bloom è composto da un array di $2^b = m$ elementi in cui ogni elemento è binario: ogni cella dell'array può valere solo zero o uno. Inizialmente, l'array è impostato a zero su tutte le sue celle.

Un filtro di Bloom usa poi k funzioni di hash indipendenti tra loro che hanno questa forma:

$$h : \{0,1\}^* \to n,\ 0 \leq n < m$$

Quando un nodo riceve un pacchetto, ne estraggo una stringa su cui applico tutte le funzioni di hash del filtro. Per ogni output, imposto a uno l'indice corrispondente nell'array.

> [!note] Nota Quando il filtro si riempie troppo, diventerà inefficiente e dovrà essere resettato.

Ora, se voglio scoprire se un pacchetto è transitato per un dato nodo, ne calcolo gli hash:

- Se ad ogni posizione dell'array trovata tramite gli hash c'è un uno, posso concludere che il pacchetto è passato da quel nodo
- Se in almeno una posizione dell'array trovata tramite gli hash c'è uno zero, il pacchetto non è transitato per quel nodo

Questo algoritmo non può dare falsi negativi, ma può dare falsi positivi. Infatti, se uno o più pacchetti collidono negli stessi campi dell'array di un altro pacchetto, potrei vedere tutti uno quando, in realtà, quel pacchetto non è mai transitato.

La probabilità di ricevere un falso positivo da questo algoritmo cresce man mano che si aggiungono elementi all'array. Proviamo a calcolarla:

- Sappiamo che la funzione hash h ha un output di lunghezza b bit
- Sappiamo che utilizziamo k funzioni hash diverse
- Sappiamo che abbiamo inserito n elementi nel filtro

[5] Un esempio è il filtraggio degli URL nei browser per bloccare i siti di phishing.
[6] Si tratta di spoofing.
[7] Non è detto che un pacchetto passi per gli stessi nodi tra sorgente e destinazione e viceversa.

[!question] Qual è la probabilità che, dopo l'inserimento di un pacchetto tramite l'uso di una sola funzione hash, un bit dell'array valga uno?

$$P_1 = \frac{1}{m}$$

[!question] Qual è la probabilità che, dopo n inserimenti, un elemento dell'array sia ancora a zero?

$$P = \left(1 - \frac{1}{m}\right)^{kn} = e^{-kn/m}$$

[!question] Qual è la probabilità di trovare un falso positivo con l'algoritmo di Bloom?

$$(1 - P)^k = \left(1 - e^{-kn/m}\right)^k$$

Quindi:

- Se la dimensione dell'array aumenta, i falsi negativi diminuiscono
- Se il numero di elementi inseriti aumenta, i falsi negativi aumentano
- Se il numero di funzioni hash aumenta, il numero di falsi positivi aumenta e diminuisce allo stesso tempo, dato che è sia nel numeratore della frazione nell'esponenziale, sia all'esponente dell'espressione intera

Dobbiamo quindi trovare un valore ottimo di k. Lo facciamo derivando l'espressione trovata ed eguagliandola a zero. Troviamo:

$$k = \frac{m}{n} \log 2$$

Che dà il minimo numero di falsi positivi pari a $0.62^{m/n}$.

Per avere un rate di falsi positivi δ, allora, è necessario un array di dimensioni:

$$m = -1.44n \log_2 \delta$$